Object Oriented Programming for Scientists

Tom Clune

SIVO Fortran 2003 Series

April 22, 2008

Logistics

- Materials for this series can be found at http://modelingguru.nasa.gov/clearspace/docs/DOC-1375
  - Contains slides and source code examples.
  - Latest materials may only be ready at-the-last-minute.
- Please be courteous:
  - Remote attendees should use “*6” to toggle the mute. This will minimize background noise for other attendees.

Outline

- Weaknesses of structured programming
- Detailed motivating example
- Basic concepts of OOP
- Applying OOP to motivating example
- Extents of applicability

Caveats

- OOP is a major paradigm shift which generally takes years to fully absorb.
- This talk is meant to *motivate* the rationale for using OOP in some circumstances within scientific models.
  - This talk is ***not*** meant as a substitute for actual training/experience.
  - Lots of excellent sources on the web.
  - Most examples are motivated by computer science considerations and may therefore be unconvincing for typical physical scientists.

(Narrow) History of OOP

- OOP grew out of perceived weaknesses/difficulties of structured programming:
  - Structured programs consisted of (global) data structures and disjoint procedures for accessing/modifying the data structures.
  - Difficulties arise especially for *large* systems composed in this manner.
- Weakness 1: Lack of support for ***encapsulation***
  - Modifications are difficult/expensive
    - Explicit references to data structure components forces frequent and pervasive changes on implementation as the data structure evolves over time.
    - Example: “Y2K” bug. Representation was explicit throughout the code.
  - Developers need to be expert in all parts of the application.
  - Limited modularity
    - **DRY** principle: **D**on’t **R**epeat **Y**ourself

History (cont'd)

- Weakness 2: Lack of support for ***extension/inheritance***
  - Isolated use cases that require different logic cannot be directly supported. Workarounds are tedious at best and tend to bloat logic and data structures.
  - Weakness 2b: Centralized development constraint
    - If an external developer creates a useful extension, she must push the extension back to the original developers in order to be of use to other users.
    - Common problem for developers of infrastructure layers.
    - E.g. if I create a new type of grid for ESMF, I cannot share the extension with other users in any simple manner. Instead, ESMF core development would need to incorporate the extension in later releases.

History of OOP (cont'd)

- Weakness 3: Lack of support for ***polymorphism***
  - Sometimes referred to as ***dynamic dispatch***
  - Common scenarios involve multiple implementations of the same functionality.   Support for variations leads to pervasive nested conditionals which increase complexity and errors.
  - Examples:
    - Support for multiple coordinate systems or grids
    - Support for multiple nonlinear solvers
- Weakness 4: Lack of support for ***templates***
  - Developers often encounter the need to support several data structures that are nearly identical but vary in some systematic ways.
  - Difficult to maintain consistency as such structures are extended.
  - E.g. real and integer arrays

Motivating Example

- Suppose we have an algorithm which involves a system of linear equations at some intermediate stage:

  $$A\,x = b$$

  - Initially we create a procedure that looks like:

    ```
    subroutine matrixSolve(array, rhs, solution)
    ```

    and declare local variables:

    ```
    real :: matrix(n,n)
    real :: solution(n), rhs(n)
    ```

  - Later development shows that the same equation must be solved multiple times for the same rhs. So we use LU decomposition for performance and have two procedures:

    ```
    subroutine LUFactor(array, LUfact, pivots)
    subroutine LUSolve(LUfact, pivots, rhs, solution)
    ```

    and local variables:

    ```
    real :: LUFactorization(n,n)
    integer :: pivots(n)
    ```

Example 1 (cont'd)

- Notice how our algorithm is already exposing aspects of matrix solving that are irrelevant to the algorithm
  - Local variables (pivot, LU factorization)
  - Methods: factor, LU backsubstitution
  - If we change the linear solver, we will probably have to change our driver code for the solver.
  - In real world cases, the "hardwiring" of the solver might occur frequently throughout the application.

Example 1 (cont'd)

- Now we discover that many (but not all) cases actually involve large banded matrices, and we want to save space and time for those:
  - Local variables

```
logical :: isBanded
integer :: nUpperBands, nLowerBands
real, allocatable :: bandedMatrix(:,:)
real, allocatable :: bandedFactors(:,:)
```

  - And conditionals:

```
if (isBanded) then
        call bandedLUFactor()
else
        call LUFactor()
end if
…
if (isBanded) then
        call bandedLUSolve(…)
else
        call LUSolve(…)
end if
```

Example

- Variation in our linear solver is starting to significantly pollute our high-level algorithm
  - More local variables
    - Many not even used in any given invocation
  - Lots of conditionals
    - Code bloat
    - Extra complexity.
- But wait … it can get worse!

Example (cont'd)

- Years later, the size of our matrices has grown considerably due to increased model resolution/data
- Analysis of our algorithm shows that in many (but not all) cases, an iterative solution would converge quickly to sufficient accuracy.
  - A variety of *preconditioners* are available, but we're not sure which will work best in practice.
- Further analysis shows that even in some parameter regimes, many matrix elements are approximately 0. Optimization is obtained by using a compressed sparse matrix representation.

Example (cont'd)

- Local Variables

```
logical :: useIteration
logical :: isSparse
real, allocatable :: sparsePreconditioner(:,:)
real, allocatable :: bandedPreconditioner(:,:)
real, allocatable :: sparseMatrix(:)
integer, allocatable :: sparseindex(:)
```

- Logic:

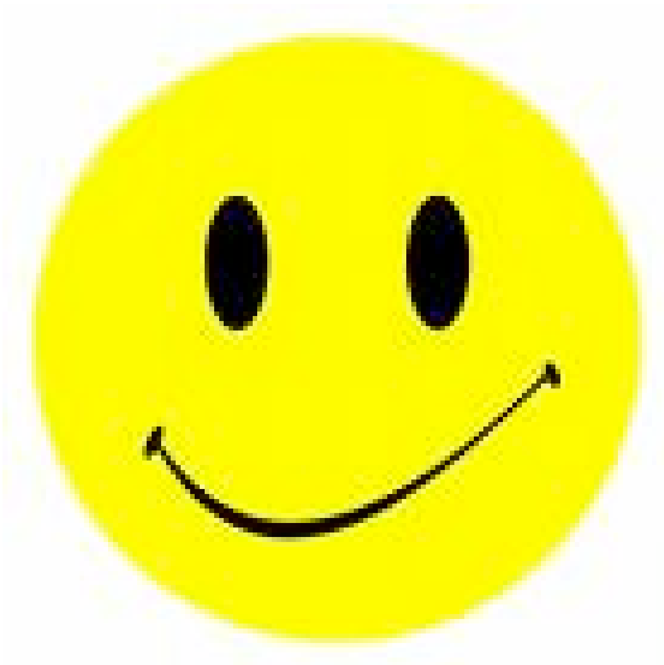
```
if (isSparse) then ! Always use iterative
        call factorPreconditioner(…)
elseif (isBanded) then
        if (useIteration) then
                call factorBandPrecond(…)
        else
                call bandLUFactor(…)
        endif
else ! Full matrix
        if (useIteration) then
                call factorFullPrecond(…)
        else
                call LUFactor(…)
        end if
end if
```

Example 1 (cont'd)

- Now suppose that someone decides to allow for iterative methods for the solution of the matrices.
  - Need to allow for preconditioners
  - Need an initial "guess"
  - Need to allow for convergence tests
  - Need to allow for variations on iterative approach
- All of this would actually be somewhat more messy than I have indicated here.
- ***What has happened!? The algorithm we are working with just needs to solve a system of linear equations!***
  - If multiple parts of our program need to solve matrices they may also be subject to the same escalation in complexity.
  - Question: Can't we somehow "hide" the complexity elsewhere in the software? Exposing only the commonalities at the top level?

Example 1 (cont'd)

- **And now … we need it to work in parallel on a cluster!**



Job security for life.

Other examples

- Air parcel trajectory code
  - Needs to support multiple vector fields
    - Analytic
    - File-based
      - Multiple interpolation schemes
  - Needs to support multiple integration schemes
    - Runge-Kutta (2nd, 4th, 8th order)
    - Adams-Bashforth, etc.
  - Can we hide details of spherical coordinates from other layers?
- Parallelization
  - Can we write our algorithms such that they appear serial?

Other examples

- Multiple Computational Grids
  - E.g. for coupled Earth systems we might have
    - Lat-Lon (Arakawa A, B, C, D)
    - Cubed-Sphere (Arakawa …)
    - Icosahedral
  - Some subsystems can “work” with any grid, while others are dependent on specific representations.
  - Coupling can require custom interpolations between grids.
  - Can we provide a software layer that supports various grid-specific operations while hiding the details from the layers that don’t really care which grid is being used?
    - Domain-decomposition, halo-fill
    - I/O operations

What is OOP

- Object oriented programming is a paradigm in which the fundamental participants are “objects” which embody both *state* and *behavior.*
  - A ***class*** is a set of properties and related procedures which access/modify those properties.
  - ***Objects*** are individual instances of classes.
    - State of an objects consists of the values of the class properties.
    - Behavior of objects is expressed in terms of ***methods*** which are the class procedures. Methods have privileged access to object state.
    - Method invocation may look different than regular procedure calls.
- Within a program, objects interact with each other by sending messages (i.e. invoking methods)
- A not-so-obvious example of a class is that of Fortran arrays:
  - Methods include shape(), size(), transpose(), minval(), etc.

Encapsulation

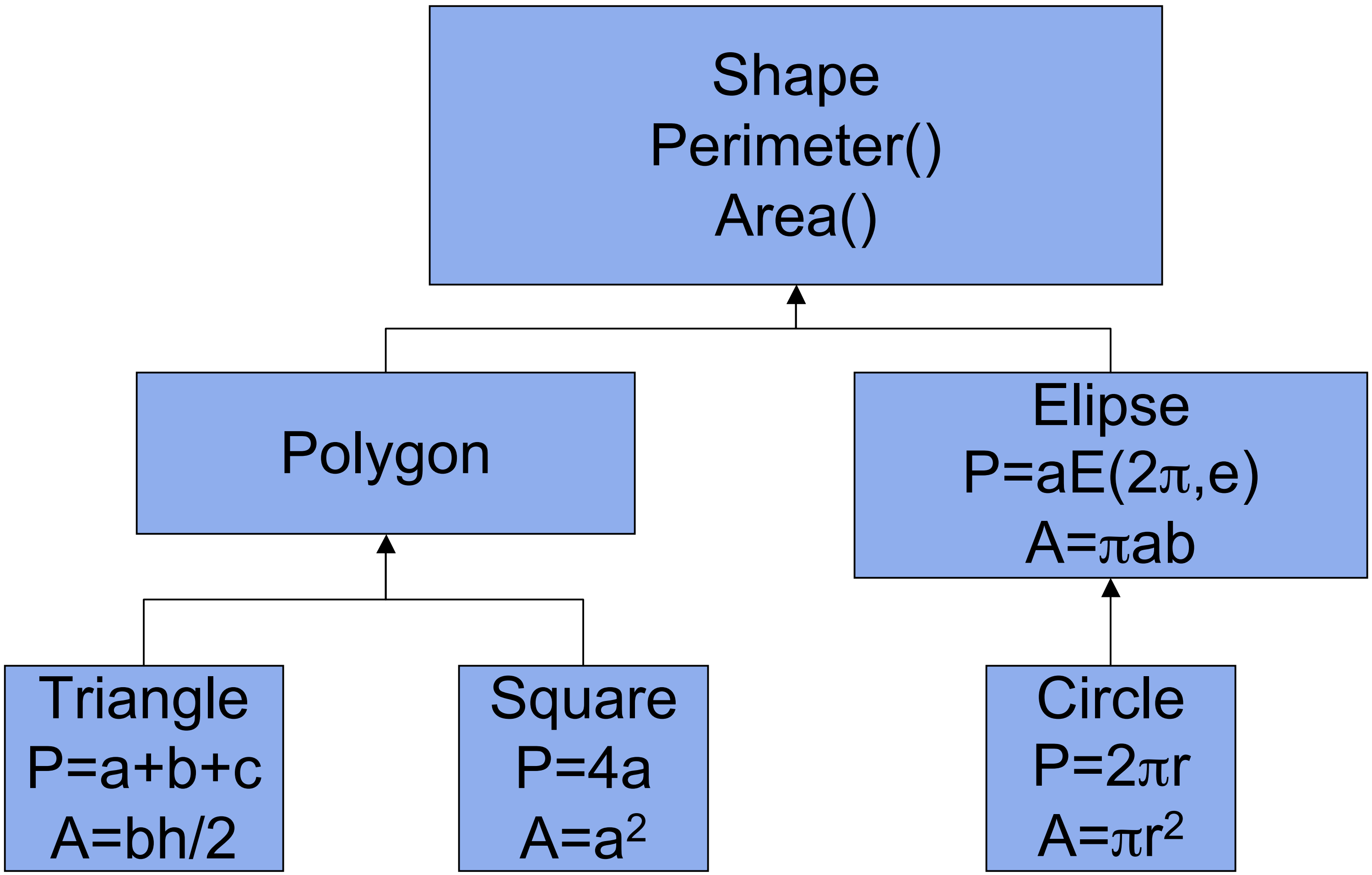
- **Encapsulation** is the ability to isolate and hide implementation details within a software subsystem.
  - Instead of directly accessing items in a data structure, *methods* are invoked to retrieve/modify.
  - If implementation details change, access *methods* are updated and client code remains unchanged.
  - E.g.

    ```
      month = date % month ! Assumes “month” field
    becomes
       month = getMonth(date) ! Does not assume “month”
    ```
  - Remember - the big wins are for complex software with many complex data structures.
- Note: Fortran 90 introduced strong encapsulation capabilities with public/private access for module entities.

Inheritance

- ***Inheritance*** is a way to form new classes using classes that have already been defined.
  - Original class is referred to as the ***base*** class (or ***parent*** class)
  - New class is referred to as the ***child*** class or ***subclass***
  - Intent is to reuse significant portions of base class.
    - Child class may add additional fields/components
    - Child class may ***override*** some methods of the parent class and leave other behaviors unchanged.
- Inheritance relations always form hierarchical trees.
- Fortran 2003 introduces inheritance (keyword: **`extends`**)
- **Child class should be usable in *<u>any</u>* context where the base class is usable.**
  - Useful notion: "is-a" relationship categorization:
    - E.g. frog is-a kind of amphibian
    - Sparse matrix is-a kind of matrix

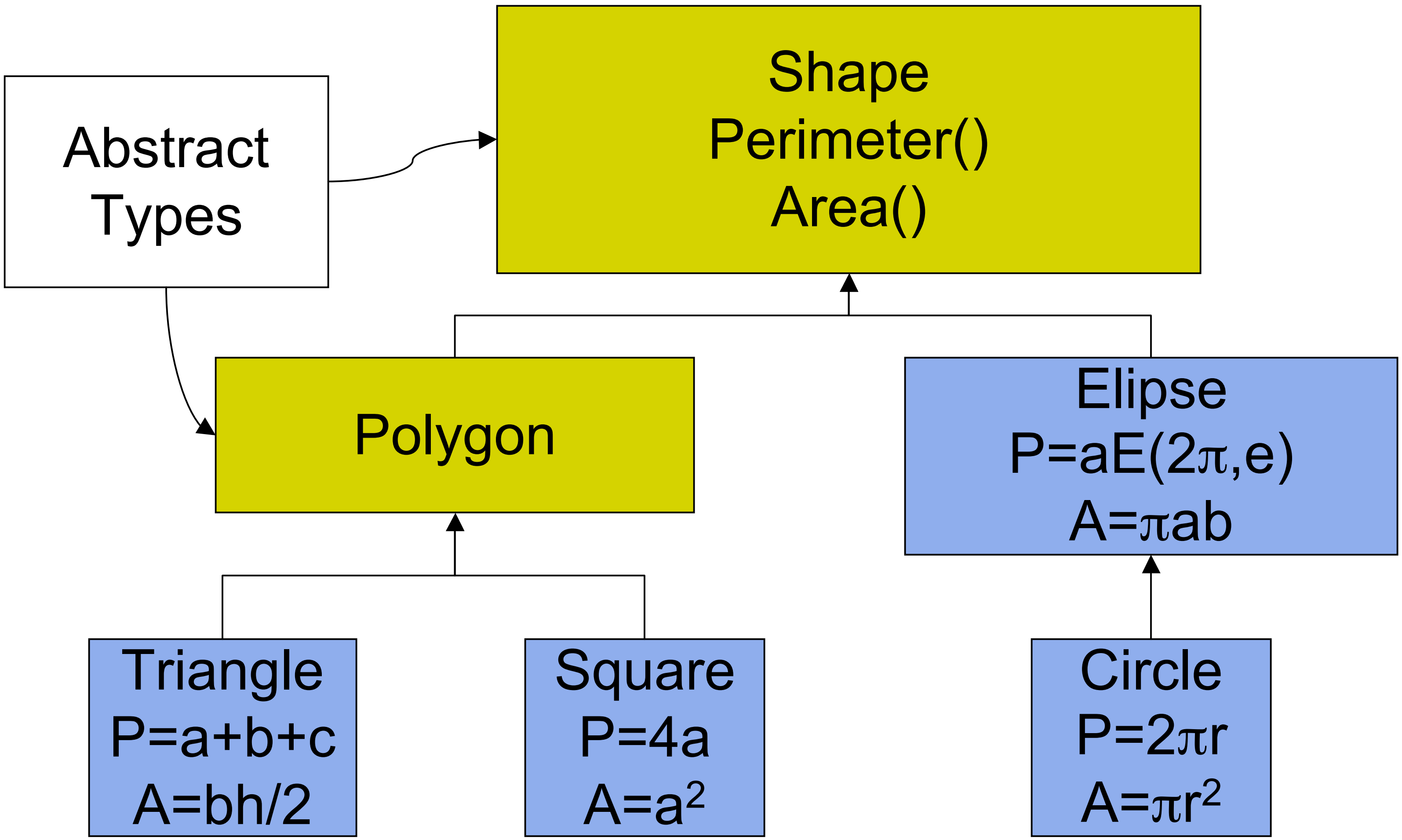
Inheritance Example



Inheritance (cont.d)

- *Inheritance Pitfall* - the real world is not always easily divided into neat categories:
  - Obligatory example: the platypus (an egg-laying mammal)
  - Subtle conflicts can ruin an OO design
- ***Abstract*** and ***Concrete*** classes
  - A common scenario in OOP is for multiple variations to exist without any particular base implementation from which to inherit.
  - The solution is to use an ***abstract*** class which defines the shared interfaces but *defers* the implementation to the subclasses.
  - Subclasses are referred to as ***concrete*** classes.
  - Cannot declare objects of the abstract class; only of concrete classes.
  - Examples:
    - Grid - no generic kind of grid just lots of subclasses.
    - AtmosphericGCM could be abstract, with concrete implementations for GEOS5_AGCM and GISS_AGCM. Encourages plug-and-play.

Inheritance Example



Function/procedure pointers

- While not strictly an OO concept, function pointers are a major part of the implementation of OO abstractions.
  - A function pointer is a data type that is able to be associated with actual functions/procedures. The association is determined at *run-time*.
  - Data structure with function pointer can be used to invoke different behavior in different contexts by associating with different actual functions.
  - No analog in Fortran 95 - but introduced in Fortran 2003
    - ***Not*** simply function dummy arguments - no way to save

Polymorphism

- ***Polymorphism*** is the capability of treating objects of a subclass as though they were members of the parent class.
- A ***polymorphic variable*** is one whose actual type is not known at compile time.
  - Run-time environment calls the appropriate methods on depending on actual type (or ***dynamic*** type)
    - Implemented with ***dynamic binding*** (usually function pointers)
  - Details of associating with specific type are language dependent
- Polymorphism and inheritance are distinct aspects but are typically applied together for maximum impact.
- E.g. polymorphic variable myShape of class “Shape” will compute the compute area/perimeter according to type set at run time.
- Introduced in Fortran 2003.

Advantages of Polymorphism

- *Generic* programming - high level algorithms are written in terms of the base class. Do not need to write variants for each subclass.
  - E.g. an algorithm working with linear equations can be written in terms of methods for generic matrices, while the specific operations (factor(), solve()) are implemented differently for the subclasses (Dense, Sparse, Banded)
- Allows customization *<u>without</u>* violating encapsulation.
  - Extension does not require access to source of the base-class.
  - Rare case where one can eat-the-cake and have-it-too.

Aside on Overloading

- AKA ***ad-hoc*** polymorphism
- Ability to use the same name for multiple procedures.
  - Actual procedure used is determined by type of arguments.
- ***Not*** based upon any type hierarchy
  - No reuse is possible - each type must have a full implementation of the overloaded procedure.
- Introduced in Fortran 90 with interface blocks

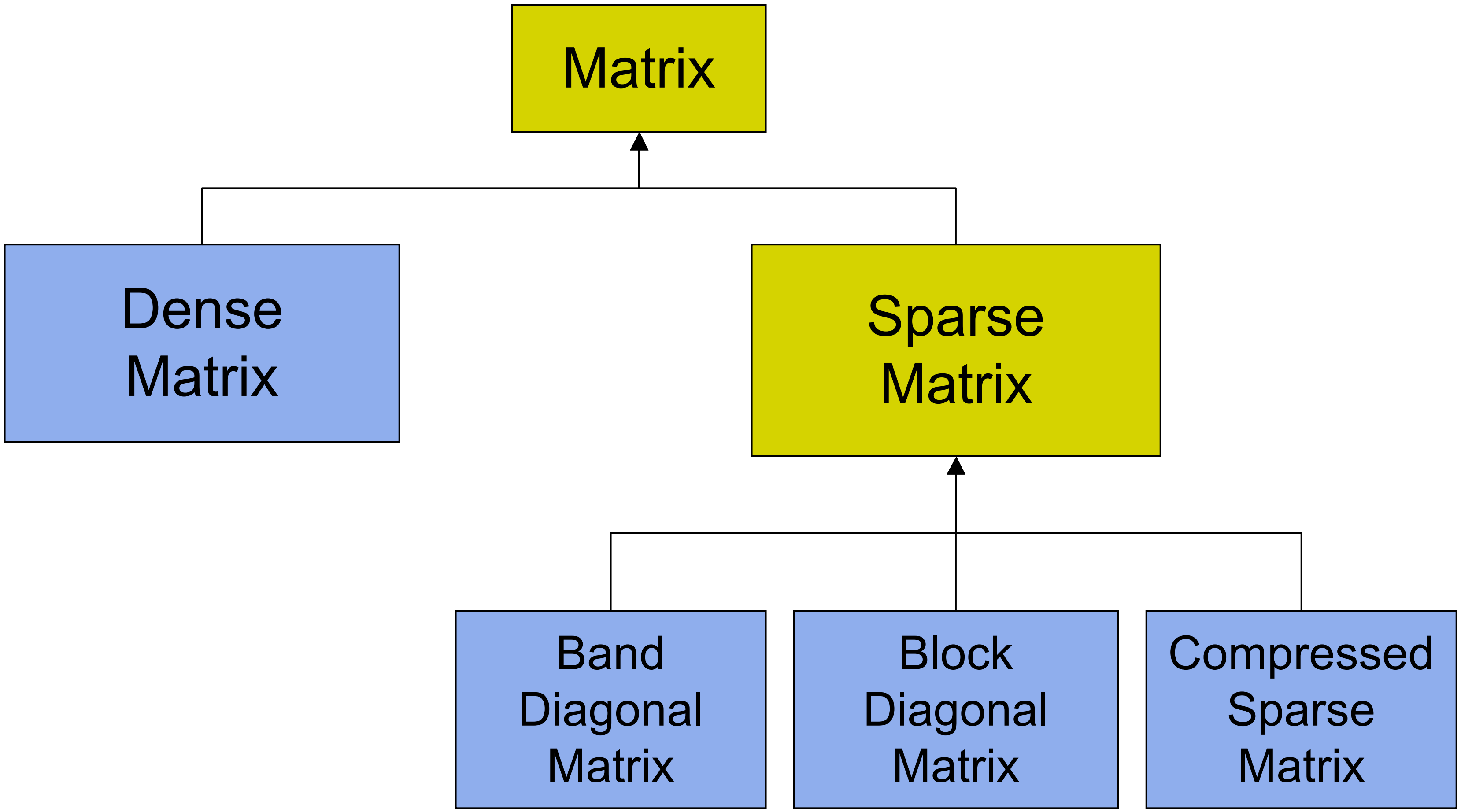
Templates

- AKA ***Parametric Polymorphism***
- Some languages support the ability to declare multiple similar classes simultaneously.
  - Routines using the type then specify which case to use
  - Distinct from first notion of polymorphism
    - Can have performance advantages - static binding
    - Not generally as flexible
- Fortran 2003 introduces a limited form
  - Derived types can be parameterized for “kinds” and sizes.
  - <u>*Cannot*</u> parameterize integers and reals simultaneously.

Example 1 revisited

- Using OOP terminology we can now sketch out a design which is more modular.
- First, we want to support different internal representations of matrices, and introduce an abstract class: **Matrix**
  - Subclass **DenseMatrix** would use conventional array storage
  - Subclass **SparseMatrix** would contain
    - **BandedMatrix**
    - **BlockDiagonalMatrix**
    - **CompressedSparseMatrix**
- Fundamental methods could be
  - Get matrix element I,J
  - Matrix-vector multiplication - needed for iterative solvers
  - Row operations ($row_i = row_i + x * row_j$) needed by direct solvers
  - Perhaps use stubs for combinations we don't want to support. (E.g. probably don't need direct solve on CompressedSparseMatrix)

Matrix Class Hierarchy



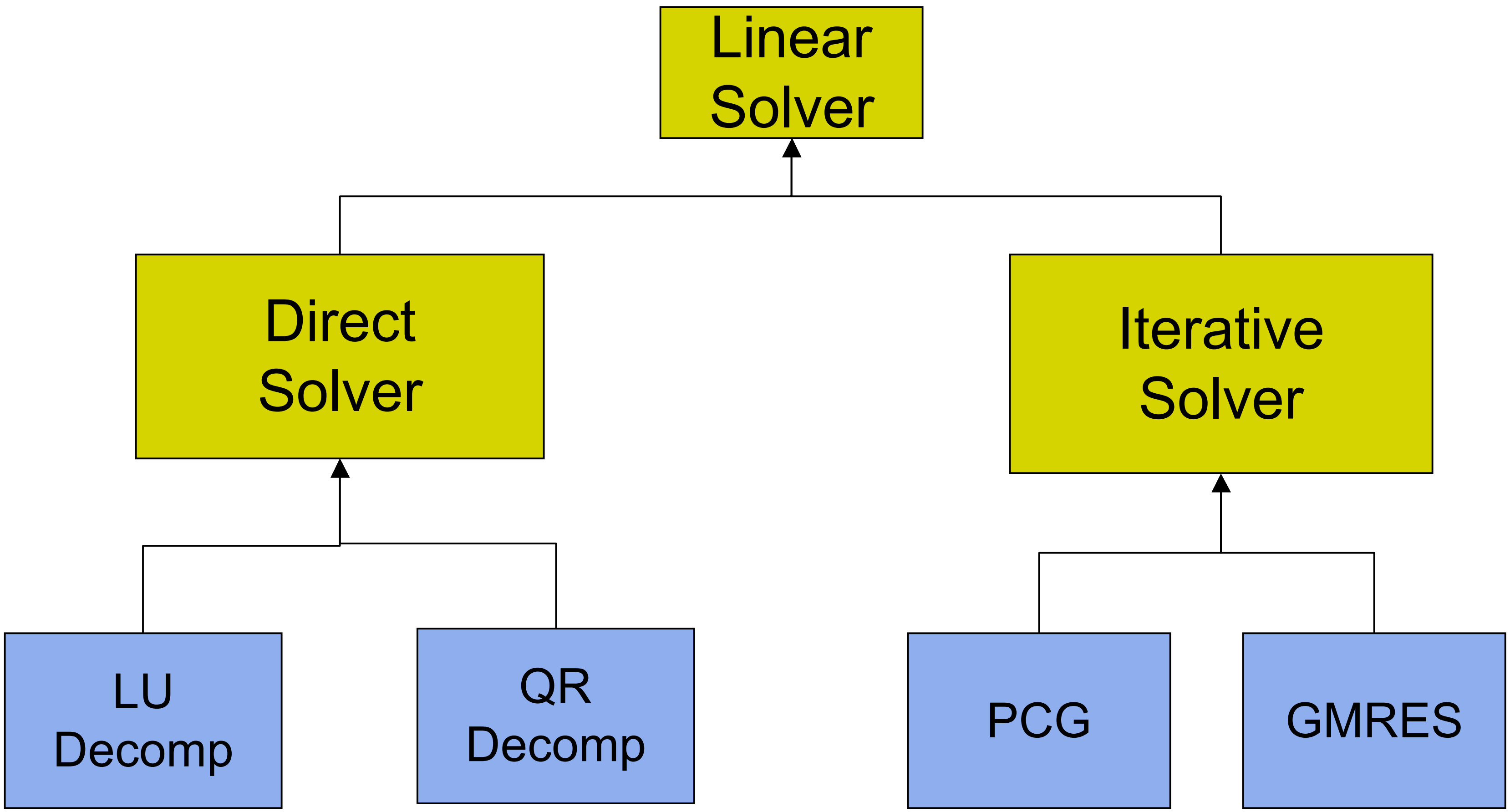
Example 1 revisited

- For the solver hierarchy we have the class: **MatrixSolver**
  - Abstract since we will have different representations of the underlying matrix and no default representation:
  - Primary methods are *preprocess*() and *solve*()
    - *preprocess*() would do any initial calculations such as factorization that would be used for multiple solve() operations.
    - *solve*() would accept a rhs and return a solution
- Note that the hierarchy should make <u>*no*</u> assumption about underlying implementation of matrices.
  - Just rely on methods from the Matrix base class.
  - In practice we may violate this somewhat for performance reasons, esp. in the case of the direct solver. Modest retreat in struggle against complexity.

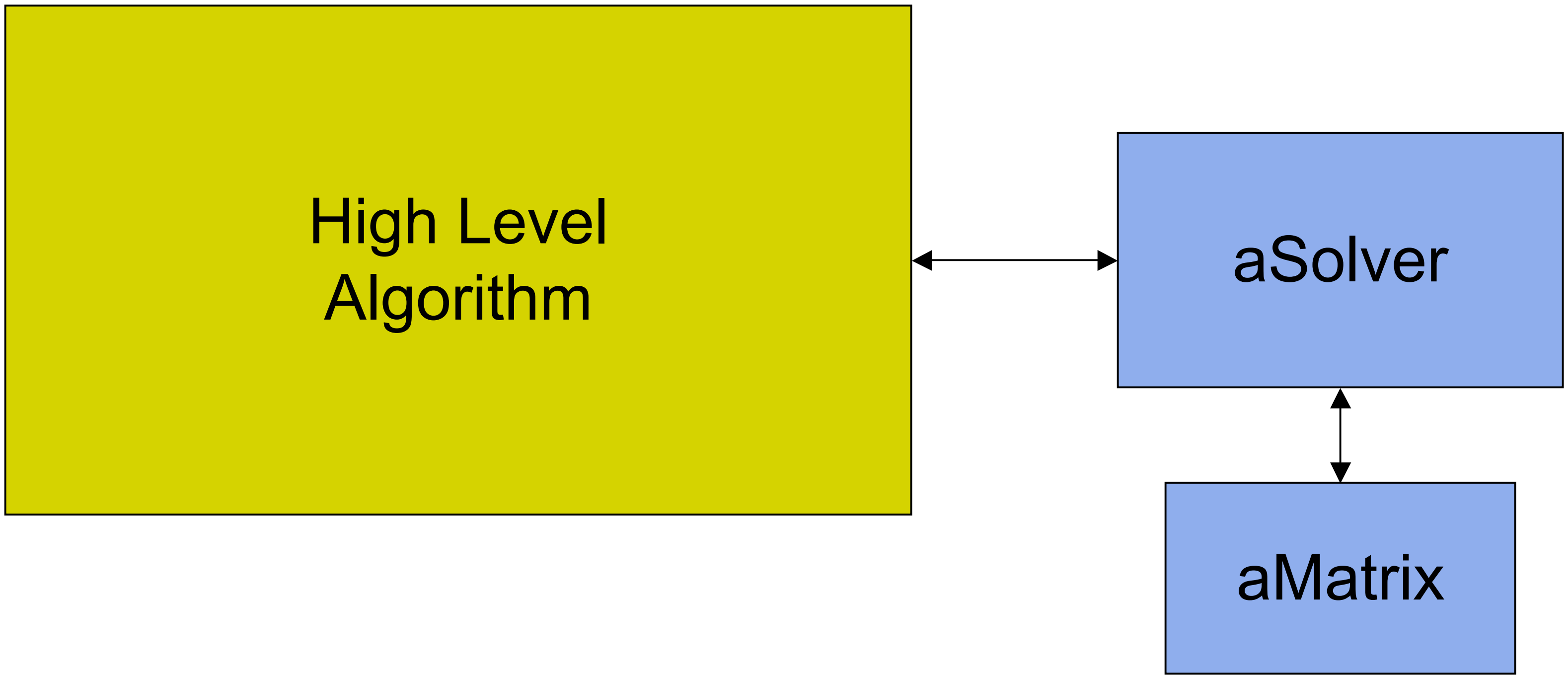
Example 1 cont'd

- Subclasses:
  - **DirectMatrixSolver**
    - **LU_MatrixSolver**
    - **QR_MatrixSolver**
  - **IterativeMatrixSolver**
    - **PCG**
    - **GMRES**
    - Iterative solvers would optionally accept a preconditioner and a tolerance.
    - **Preconditioner could itself be a MatrixSolver object!**

Linear Solver Hierarchy



Using the linear solver



Algorithm "has-a" MatrixSolver initialized with a Matrix object. Subtypes of each are not directly known. Matrix and MatrixSolver classes *collaborate*.

OOP and Model Infrastructure

- The clearest case for OOP in scientific models is in the “infrastructure” which manages the various model abstractions.
  - Infrastructure includes
    - I/O
    - Computational grid
    - Loop constructs
    - Domain decomposition
    - Calendars/clocks
  - Common infrastructure issues among various Earth system models led to the creation of the ESMF. While not truly OO, ESMF is strongly encapsulated and has an object based look-and-feel.
    - With the availability of OOP, some aspects of ESMF become trivial, and others could be extended to be far more powerful.

OOP and Numerics

- As seen in the earlier example, OOP can be a useful approach for some numerical issues.   When multiple data representations are possible and require different (but comparable) algorithmic treatments, inheritance/polymorphism become very important.

Parameterized physics?

- Even when the the detailed implementation of a parameterized model is not based upon objects, it might make sense to consider the model to be a concrete implementation of some abstract model.
  - A strong step towards enabling plug-and-play with other implementations
  - Encourages user extensions/enhancements and eases the reintegration of such changes into the original model.

Resources

- **SIVO Fortran 2003 series:** https://modelingguru.nasa.gov/clearspace/docs/DOC-1390
- **Questions to Modeling Guru:** https://modelingguru.nasa.gov
- **SIVO code examples on Modeling Guru**
- **Fortran 2003 standard:** http://www.open-std.org/jtc1/sc22/open/n3661.pdf
- ***John Reid summary:***
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz
- ***Newsgroups***
  - http://groups.google.com/group/comp.lang.fortran
- ***Mailing list***
  - http://www.jiscmail.ac.uk/lists/comp-fortran-90.html

Next Fortran 2003 Session

- Inheritance in Fortran 2003
- Tom Clune will present
- Tuesday, May 06, 2008
- B28-E210 @ 12:00 noon